PL/SQL

Proč používat PL/SQL?

- Samotné SQL neumožňuje používat procedurální kód (napsat si vlastní řízení příkazů – podmínky, cykly aj.), což může být omezující (pomocí SQL nejsme schopni naprogramovat složitější databázové algoritmy, pomocí PL/SQL ano).
- Často používané operace je možné zapouzdřit do procedur/funkcí, které se časově efektivně vykonávají.
- Obecně je jazyk PL/SQL neopodstatněně málo využíván. Jeho využití znamená navýšení výkonu, modularizaci a snadnou centrální správu procedurálního kódu a bohužel vázání se na konkrétní databázový systém (jediný podstatnější zápor).
- Oracle umožňuje psát procedury/funkce nejen v PL/SQL, ale také v jazycích C a Java.

Srovnání SQL, PL/SQL a Java / Python:

neproceduralní:	procedurální:	objektové:
SQL	PL/SQL	Java Python

flexibilita:

SQL < PL/SQL < Java / Python

efektivita (výkon):

SQL > PL/SQL > Java / Python

Co PL/SQL nabízí?

- Deklaraci konstant, proměnných, kurzorů pro průchod výsledků SELECTu,
- podporu transakčního zpracování,
- ošetření chybových stavů pomocí výjimek,
- modularitu – procedury, funkce je možné zařazovat do balíčků, v jedné funkci je možné využít jinou funkci nebo proceduru...
- podporu dědičnosti,
- ...

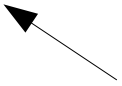
Proč zrovna PL/SQL a ne třeba C, Javu?

- PL/SQL je výkonný jazyk pro psaní procedurálního kódu (procedur, funkcí, triggerů):
  - Používá přímo datové typy Oracle, nejsou potřeba žádné konverze, je těsně navázán na SQL a objekty v dané databázi (efektivní kontrola),
  - používané datové typy lze deklarovat obecně (např. nadeklarovat proměnnou stejného typu jako je určitý sloupec tabulky), tak, aby se kód nemusel měnit při menších změnách tabulek,
  - jeho použití je snadnější než použití C/Javy (např. otevření a uzavření dotazu je prováděno automaticky),
  - dotaz v rámci funkce (procedury) je analyzován pouze jednou při kompilaci funkce (procedury), poté je už jen efektivně vykonáván.
- PL/SQL by se mělo používat až tehdy, když si nevystačíme s prostým SQL dotazem (přitom syntaxe SELECTů je bohatá...).

Základní struktura procedurálního kódu v PL/SQL

Kód jde psát přímo jako nepojmenovanou proceduru (anonymní blok):

```
DECLARE
   nepovinná deklarační sekce pro proměnné,
   konstanty, kurzory
BEGIN
   povinná výkonná sekce, jde sem případně
   zanořit i další anonymní blok...
EXCEPTION
   nepovinná sekce pro zpracování výjimek
END;
/ nakonec lomítko pro kompilaci kódu
```



Nutné v SQL*Plus, nikoli v SQL Developeru

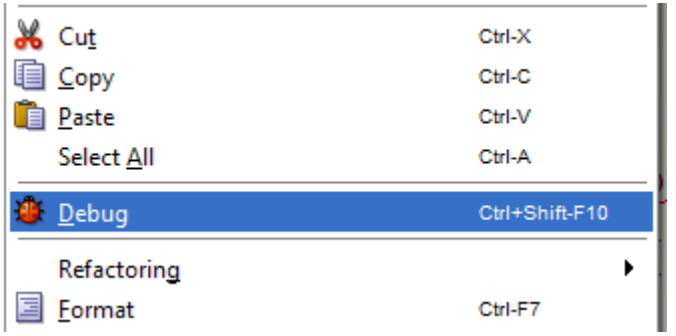
Hello World příklad

Nutné v SQL*Plus, nikoli v SQL Developeru

```
-- nejprve zapnout textový výstup:
SET SERVEROUT ON

-- poté vlastní kód:
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello World');
   -- použití procedury PUT_LINE
   -- z balíčku DBMS_OUTPUT
END;
/
```

Pro spuštění kódu a zobrazení informací vypisovaných na textový výstup spusťte příkaz v Debug režimu:
Ve worksheetu klikněte na pravé tlačítko a vyberte Debug):



Proměnné v PL/SQL

Před prvním použitím ve výkonné sekci je potřeba proměnnou nadeklarovat v deklarační sekci:

```
DECLARE
  v_promenna1 NUMBER(3);
  v_promenna2 NUMBER NOT NULL DEFAULT 88;
  v_promenna3 NUMBER := 77;
  -- typ proměnné podle jiné proměnné:
  v_promenna4 v_promenna3%TYPE;
  -- typ proměnné podle sloupce tabulky:
  v_promenna5 zamestnanec.jmeno%TYPE;
```

Komentáře v PL/SQL

```
/* Víceřádkový

komentář */


-- jednořádkový komentář
```

- Jednořádkové komentáře používat od začátku řádku (na samostatném řádku)!

Řízení toku programu – větvení

```
IF podmínka1 THEN
   posloupnost_příkazů1
ELSIF podmínka2 THEN
   posloupnost_příkazů2
ELSE
   posloupnost_příkazů3
END IF;
-- větve elseif (může jich být více) a else
-- jsou nepovinné
```

Řízení toku programu – větvení

```
CASE
   WHEN podmínka1 THEN  posloupnost_příkazů1;
   WHEN podmínka2 THEN  posloupnost_příkazů2;
   ..
   WHEN podmínkaN THEN  posloupnost_příkazůN;
   [ ELSE posloupnost_příkazůN+1; ]
END CASE;
```

- V podmínce může být např. v_promenna BETWEEN 1 AND 5 pro vyčlenění intervalu hodnot nějaké proměnné.

Řízení toku programu – cykly

- Jednoduchý cyklus LOOP:

```
LOOP
  posloupnost_příkazů
IF podmínka THEN
  EXIT;
END IF;
END LOOP;

  nebo...

LOOP
  posloupnost_příkazů
  EXIT WHEN podmínka;
END LOOP;
```

Řízení toku programu – cykly

- Cyklus FOR s čítačem:

```
FOR počítadlo IN [REVERSE] nejnižší hodnota..nejvyšší
LOOP
  posloupnost_příkazů
END LOOP;
```

např.:

```
FOR v_citac IN 1..10
LOOP
  DBMS_OUTPUT.PUT_LINE('v_citac = ' || v_citac);
END LOOP;
```

Řízení toku programu – cykly

- Cyklus WHILE s podmínkou na začátku:

```
WHILE podmínka
LOOP
  posloupnost_příkazů
END LOOP;
```

Naplnění proměnných SELECTem

```
SELECT [* | seznam_atributů]
INTO [seznam_proměnných nebo proměnná typu záznam]
FROM název_tabulky
WHERE podmínky_výběru
```

- Je potřeba, aby SELECT vracel právě jeden záznam, jinak by se vyvolala výjimka.

Naplnění proměnných SELECTem – příklad

```
DECLARE
  v_jmeno zamestnanec.jmeno%TYPE;
  v_id zamestnanec.zamestnanec_id%TYPE;
BEGIN
  SELECT jmeno, zamestnanec_id INTO v_jmeno, v_id
  FROM zamestnanec WHERE zamestnanec_id = 6;

  DBMS_OUTPUT.PUT_LINE('Jméno: ' || v_jmeno);
  DBMS_OUTPUT.PUT_LINE('Id: ' || v_id);
END;
/
```

Naplnění proměnných s ošetřením výjimek

```
DECLARE
  v_jmeno zamestnanec.jmeno%TYPE;
  v_id zamestnanec.zamestnanec_id%TYPE;
BEGIN
  SELECT jmeno, zamestnanec_id INTO v_jmeno, v_id
  FROM zamestnanec WHERE zamestnanec_id = 6;
  DBMS_OUTPUT.PUT_LINE('Jméno: ' || v_jmeno);
  DBMS_OUTPUT.PUT_LINE('Id: ' || v_id);
EXCEPTION
  -- ošetření výjimky při nenalezení dat
  WHEN NO_DATA_FOUND
  THEN DBMS_OUTPUT.PUT_LINE('Data nenalezena');
  -- ošetření výjimky při nalezení více řádků
  WHEN TOO_MANY_ROWS
  THEN DBMS_OUTPUT.PUT_LINE('SELECT vybral mnoho řádků, ne jeden!');
END;
/
```

Kurzory

- **Kurzor** je privátní pracovní oblast, kterou databázový server vytvoří pro každý SELECT – je spojen s dotazem typu SELECT a určen pro průchod výsledky. Kurzor lze vytvářet:
  - **Implicitně** – automaticky databázovým serverem, programátor se o něj sám nestará,
  - **explicitně** – nadeklaruje jej a vytvoří přímo programátor.
- Základní kroky při práci s kurzorem:
  - Deklarace kurzoru (CURSOR - IS),
  - otevření kurzoru (OPEN),
  - výběr dat prostřednictvím kurzoru (FETCH - INTO),
  - uzavření kurzoru (CLOSE).

Kurzory – syntaxe základních kroků

```
-- V deklaračním bloku:
CURSOR <název kurzoru> IS <příkaz SELECT>;
-- Dále ve výkonném bloku:
OPEN <název kurzoru>;
-- Pro postupné procházení záznamů se v cyklu
-- volá:
FETCH <název kurzoru> INTO <seznam proměnných>;
-- Nakonec uzavření kurzoru:
CLOSE <název kurzoru>;
```

Stav kurzoru

- Kdykoliv můžeme otestovat stav kurzoru pomocí atributů kurzoru:
  - **<název kurzoru>%ROWCOUNT** – pořadové číslo aktuálního záznamu, pokud zatím nebyl vybrán žádný záznam, má hodnotu 0,
  - **<název kurzoru>%FOUND** – pokud poslední příkaz FETCH načetl nějaká data, má hodnotu TRUE; jinak FALSE; používá se pro zjištění konce cyklu, ve kterém se iteruje přes záznamy; obdobně funguje **<název kurzoru>%NOTFOUND**,
  - **<název kurzoru>%ISOPEN** – vrací TRUE, pokud je kurzor otevřen.

Kurzory – příklad

```
DECLARE
  v_jmeno zamestnanec.jmeno%TYPE;
  v_id zamestnanec.zamestnanec_id%TYPE;
  CURSOR k1 IS SELECT jmeno, zamestnanec_id FROM zamestnanec;
BEGIN
  OPEN k1; -- otevreni kurzoru - provedeni SELECTu
  LOOP
    FETCH k1 INTO v_jmeno, v_id;
    DBMS_OUTPUT.PUT_LINE('Jméno: ' || v_jmeno || ', Id: '
      || v_id);
    EXIT WHEN k1%NOTFOUND;
  END LOOP;
  CLOSE k1;
END;
/
```

Záznamy

- Záznam je datový typ/struktura, která zapouzdřuje více položek, i různých datových typů.
- Příklad deklarace záznamu v deklarační sekci:

```
TYPE rec_zamestnanec IS RECORD (
  jmeno zamestnanec.jmeno%TYPE;
  id zamestnanec.zamestnanec_id%TYPE;
);
```

- Nebo lze nadeklarovat záznam odpovídající řádku tabulky:

```
rec_zamestnanec zamestnanec%ROWTYPE;
```

Extrakce dat z kurzoru do záznamu

- Práce s kurzory je pomocí záznamů jednodušší: Kurzor nemusíme sami otevírat ani zavírat, ani data z kurzoru vybírat pomocí FETCH ... databázový systém se o to postará sám:

```
DECLARE
  rec_zamestnanec zamestnanec%ROWTYPE;
  CURSOR k1 IS SELECT jmeno, zamestnanec_id FROM zamestnanec;
BEGIN
  FOR rec_zamestnanec IN k1
  LOOP
    DBMS_OUTPUT.PUT_LINE('Jméno: ' || rec_zamestnanec.jmeno
      || ', Id: ' || rec_zamestnanec.zamestnanec_id);
  END LOOP;
END;
/
```

Kurzory s parametry

- Pro kurzor můžeme nadefinovat parametry, za které se dosadí konkrétní hodnoty při otevírání kurzoru. Parametry můžeme využívat v SELECTu, se kterým je kurzor svázán:

```
DECLARE
  rec_zamestnanec zamestnanec%ROWTYPE;
  CURSOR k1 (v_jmeno VARCHAR2) IS
    SELECT jmeno, zamestnanec_id FROM zamestnanec
    WHERE jmeno LIKE (v_jmeno || '%');
BEGIN
  FOR rec_zamestnanec IN k1 ('Mi')
  LOOP
    DBMS_OUTPUT.PUT_LINE('Jméno: ' || rec_zamestnanec.jmeno
      || ', Id: ' || rec_zamestnanec.zamestnanec_id);
  END LOOP;
END;
/
```

Ošetřování chyb

- V PL/SQL se mohou vyskytnout chyby:
  - **Syntaktické** – objeví se už při kompilaci,
  - **logické** – projeví se až za běhu funkce/procedury.
- Nejčastěji se vyskytují tyto vestavěné výjimky:
  - **DUP_VAL_ON_INDEX** – výskyt duplicitní hodnoty ve sloupci, kde jsou povoleny jen jedinečné hodnoty,
  - **INVALID_NUMBER** – neplatné číslo, data nelze převést na číslo,
  - **NO_DATA_FOUND** – nebyly nalezeny žádné záznamy,
  - **TOO_MANY_ROWS** – dotaz vrátil neočekávaně více než jeden záznam,
  - **VALUE_ERROR** – problém s matematickou funkcí, chybný argument,
  - **ZERO_DIVIDE** – chyba dělení nulou.

Ošetřování chyb

- Syntaxe ošetřování výjimek:

```
EXCEPTION
   WHEN <název výjimky1> THEN <příkazy1>;
   [WHEN <název výjimky2> THEN <příkazy2>; ...]
   OTHERS THEN <příkazy3>;
END;
```

- Při ošetřování výjimky můžeme použít systémové funkce SQLCODE (vrací kód chyby), SQLERRM (vrací textový popis chyby). Pro vlastní výjimky je SQLCODE rovno 1 a SQLERRM vrací text User-Defined Exception.
- Syntaxe vyvolání vestavěné nebo vlastní výjimky:

```
RAISE <název výjimky>; např.: RAISE NO_DATA_FOUND;
```

Ošetřování chyb – příklad

```
DECLARE
  v_vysledek NUMBER(9,2);
BEGIN
  v_vysledek := 5/0;
EXCEPTION
  WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE('Nastala chyba.');
   DBMS_OUTPUT.PUT_LINE('Kód chyby:' || SQLCODE);
   DBMS_OUTPUT.PUT_LINE('Popis chyby:' || SQLERRM);
END;
/
```

Použití vlastních výjimek

- Deklarace, vyvolání a ošetření vlastní výjimky:

```
DECLARE
  <název výjimky> EXCEPTION;
BEGIN
  <příkazy>;
  -- vyvolání výjimky při nějakém chybovém stavu
  -- typicky při splnění nějaké výjimky:
  IF <podmínka> THEN
    RAISE <název výjimky>;
  END IF;
EXCEPTION
  WHEN <název výjimky> THEN <příkazy>;
END;
```

RAISE_APPLICATION_ERROR

- Vyvolat výjimku (tzv. aplikační výjimku) lze také pomocí procedury RAISE_APPLICATION_ERROR(chybový kód, 'hlášení');
(z balíčku DBMS_STANDARD). Lze používat volné chybové kódy -20000 až -20999, které pak obdrží i aplikace volající uloženou proceduru/funkci. Výjimku lze zachytit ve větvi OTHERS, případně pomocí PRAGMA EXCEPTION_INIT namapovat chybový kód na vlastní jméno výjimky.
- RAISE_APPLICATION_ERROR se používá také v sekci EXCEPTION, když chceme zachycenou výjimku nechat vyvolat ven z procedury/funkce s určitým chybovým kódem a hlášením.

Procedury

- Procedura je posloupností příkazů, které jde opakovaně spouštět. Na základě vstupních parametrů jsou vráceny výsledky v podobě výstupních parametrů.
- Syntaxe:

```
CREATE [OR REPLACE] PROCEDURE <název procedury> [(<seznam parametrů>)] AS
  deklarační sekce
BEGIN
  výkonná sekce
[EXCEPTION
  sekce pro zpracování výjimek]
END;
/ -- kompilace

<seznam parametrů> = <jméno parametru1> {IN|OUT|IN OUT}
<typ> [DEFAULT <hodnota>], ...
```

Procedury – příklad

```
CREATE OR REPLACE PROCEDURE zvyseni_mzdy (procento IN NUMBER) AS
BEGIN
  UPDATE zamestnanec SET plat = plat * (1 + procento/100);
END;
/
```

Spuštění procedury 1. způsob:

```
EXECUTE zvyseni_mzdy(10);

COMMIT;
```

Spuštění procedury 2. způsob:

```
BEGIN
  zvyseni_mzdy(10);
  COMMIT;
END;
```

Poznámka 1: V SQL Developeru volejte tyto příkazy pomocí „Run Script (F5)“, aby se všechny příkazy vykonaly.

Poznámka 2: Pro potvrzení dat do databáze je nutné buď zmáčknout „Commit (F11)“,nebo uvést příkaz COMMIT jako v příkladu.

Funkce

- Funkce je prakticky to samé, co procedura, ale má explicitně určenou návratovou hodnotu.
- Syntaxe:

```
CREATE [OR REPLACE] FUNCTION <název funkce> [(<seznam parametrů>)] RETURN <datový typ výsledku> AS
  deklarační sekce
BEGIN
  výkonná sekce
  RETURN <hodnota>;
[EXCEPTION
  sekce pro zpracování výjimek]
END;
/ -- kompilace funkce
```

- Volání funkce: Funkci lze zavolat v SQL dotazu, v proceduře/jiné funkci/triggeru, tak, jako standardní funkce Oraclu.

Triggery

- Trigger („spoušť“) je procedurální kód, který lze spouštět automaticky před/namísto/po každém provedení příkazu INSERT, DELETE nebo UPDATE na vybrané tabulce. Trigger je navázán na tabulku, při zrušení tabulky příkazem DROP je zrušen i trigger.
- Syntaxe:

```
CREATE [OR REPLACE] TRIGGER <název triggeru> {BEFORE|
AFTER|INSTEAD OF} {INSERT|DELETE|UPDATE} ON <název
tabulky> [FOR EACH ROW [WHEN <podmínka>]]
[DECLARE
  deklarační sekce]
BEGIN
  výkonná sekce
[EXCEPTION
  sekce pro zpracování výjimek]
END;
/ -- kompilace triggeru
```

Triggery

- Pokud uvedeme FOR EACH ROW, trigger se vyvolá pro každý vkládaný/aktualizovaný/mazaný řádek tabulky a v těle triggeru je dostupný pseudoobjekt :NEW reprezentující nový záznam s novými hodnotami po provedení dotazu INSERT/UPDATE/DELETE a :OLD reprezentující původní záznam s původními hodnotami. Pokud se navíc jedná o BEFORE trigger, údaje v záznamu :NEW lze v triggeru měnit (modifikovat výsledné hodnoty ve sloupcích po provedení INSERT/UPDATE/DELETE). Vyvoláním výjimky ven z BEFORE triggeru lze zabránit provedení dotazu INSERT/UPDATE/DELETE.
- Pokud se neuvede FOR EACH ROW, trigger se vyvolá jen jednou před/namísto/po dotazu INSERT/UPDATE/DELETE (jednou pro celou tabulku) a v těle triggeru nebudou dostupné záznamy :NEW, :OLD.
- WHEN <podmínka> umožňuje spouštět trigger pouze při platnosti zadané podmínky.

Balíčky procedur a funkcí

- Funkce a procedury lze zapouzdřit do vlastního balíčku (pod jmenný prostor – název balíčku):

```
CREATE [OR REPLACE] PACKAGE <název balíčku> AS
  FUNCTION ... hlavička až po RETURN <typ> včetně;
  PROCEDURE ... hlavička až po seznam parametrů včetně;
  atd. ... tj. deklarace procedur a funkcí
END <název balíčku>;

CREATE [OR REPLACE] PACKAGE BODY <název balíčku> AS
  -- definice celých funkcí a procedur
END <název balíčku>;
/
```

- Spouštění funkcí/procedur: Před název funkce/procedury se předřazuje jméno balíčku:

```
JMENO_BALICKU.JMENO_FUNKCE(...);
```